CI4251 - Programación Funcional Avanzada
Tarea 2

Alessandro La corte
09-10430
<alessandroempire@gmail.com>

Mayo 8, 2015

Autómatas Finitos No Determinísticos

```
import Control.Monad
import Control.Monad.RWS
import Control.Monad.Error
import Control.Monad.State
import Control.Monad.Writer
import Control.Monad.Error
import Control.Monad.Reader
import qualified Data.Sequence as Seq
import qualified Data.Set as DS
import Data.Char
import Data.Either
import Test.QuickCheck
```

Considere el siguiente conjunto de Tipos Abstractos de Datos diseñados para representar Autómatas Finitos No-Determinísticos con $\lambda-$Transiciones ($\lambda-$NFA).

Los estados de un $\lambda-$NFA serán representados con un `newtype`. Esto tiene el doble propósito de tener enteros diferenciados, para un desempeño aceptable, y poder disponer de una instancia `Show` separada con la cual mantener la tradición de que los estados sean denotados `q0`, `q1`, ...

```
newtype NFANode = Node Int
                deriving (Eq,Ord)

instance Show NFANode where
   show (Node i) = "q" ++ show i
```

Luego, representaremos las transiciones de un $\lambda-$NFA con un tipo completo que incluye alternativas para transiciones que consumen un símbolo de entrada y para $\lambda-$transiciones. Con el tipo completo, además de la conveniencia de funciones de acceso a los campos provista por la notación de registros, también podemos tener una instancia `Show` separada que muestre las transiciones de manera más atractiva.

```
data Transition = Move   { from, to :: NFANode, sym :: Char }
                | Lambda { from, to :: NFANode }
                deriving (Eq,Ord)

instance Show Transition where
  show (Move f t i) = show f ++
```

```
                      " -" ++ show i ++ "-> " ++
                      show t
  show (Lambda f t) = show f ++ " ---> " ++ show t
```

Finalmente, aprovechando la librería `Data.Set` para representar conjuntos, podemos representar un $\lambda-$NFA de manera directa.

```
data NFA = NFA {
                   sigma   :: (DS.Set Char),
                   states  :: (DS.Set NFANode),
                   moves   :: (DS.Set Transition),
                   initial :: NFANode,
                   final   :: (DS.Set NFANode)
                 }
         deriving (Eq,Show)
```

En esta definición:

- `sigma` es el conjunto no vacío de caracteres del alfabeto.
- `states` es el conjunto no vacío de estados; siempre debe incluir al menos al estado inicial.
- `moves` es el conjunto de transiciones.
- `initial` es el estado final que *siempre* sera `q0` (o sea `(Node 0)`).
- `final` es el conjunto de estados finales.

Con estas definiciones podemos construir expresiones que denoten $\lambda-$NFA

```
nfa0 = NFA {
             sigma  = DS.fromList "ab",
             states = DS.fromList $ fmap Node [0..3],
             moves  = DS.fromList [
               Move { from = Node 0, to = Node 0, sym = 'a' },
               Move { from = Node 0, to = Node 0, sym = 'a' },
               Move { from = Node 0, to = Node 1, sym = 'a' },
               Move { from = Node 1, to = Node 2, sym = 'b' },
               Move { from = Node 2, to = Node 3, sym = 'b' }
             ],
             initial = Node 0,
             final = DS.fromList [ Node 3 ]
           }

nfa1 = NFA {
```

```
            sigma  = DS.fromList "ab",
            states = DS.fromList $ fmap Node [0..3],
            moves  = DS.fromList [
              Move { from = Node 0, to = Node 0, sym = 'b' },
              Move { from = Node 0, to = Node 1, sym = 'a' },
            Lambda { from = Node 0, to = Node 2 },
              Move { from = Node 1, to = Node 1, sym = 'a' },
              Move { from = Node 1, to = Node 1, sym = 'b' },
              Move { from = Node 1, to = Node 3, sym = 'b' },
              Move { from = Node 2, to = Node 2, sym = 'a' },
              Move { from = Node 2, to = Node 1, sym = 'b' },
            Lambda { from = Node 2, to = Node 3 },
              Move { from = Node 3, to = Node 3, sym = 'a' },
              Move { from = Node 3, to = Node 2, sym = 'b' }
            ],
            initial = Node 0,
            final = DS.fromList [ Node 4 ]
          }
```

Generando $\lambda-$NFAs

La primera parte de este ejercicio requiere que Ud. escriba las instancias `Arbitrary` que permitan generar nodos y $\lambda-$NFAs. En este sentido, queremos que la generación arbitraria sea de valores correctos.

```
instance Arbitrary NFANode where
  arbitrary = undefined --nfaNodes
      --where nfaNodes = suchThat (liftM Node arbitrary) (> 0)
```

En el caso de `NFANode` queremos que los estados siempre sean identificados con enteros *positivos*. La librería `Test.QuickCheck` incluye una forma de generar enteros positivos que Ud. debe aprovechar. Como el estado inicial `(Node 0)` *siempre* debe estar presente, no hace falta generarlo sino incluirlo manualmente en los $\lambda-$NFA.

```
instance Arbitrary NFA where
  arbitrary = undefined
```

En el caso de `NFA` queremos que el generador sólo produzca $\lambda-$NFA con estructura consistente. En este sentido, la instancia debe *garantizar*:

- Que el alfabeto sea no vacío sobre letras minúsculas.
- Que el conjunto de estados sea de tamaño arbitrario pero que *siempre* incluya a `(Node 0)`.
- Que tenga una cantidad arbitraria de transiciones. Todas las transiciones tienen que tener sentido: entre estados que están en el conjunto de estados y con un símbolo del alfabeto. Se desea que haya una $\lambda-$transición por cada cinco transiciones convencionales.
- El estado inicial siempre debe ser `(Node 0)`.
- El estado final debe ser un subconjunto del conjunto de estados, posiblemente vacío.

Es inaceptable que el generador produzca algo con la forma de un $\lambda-$NFA y luego se verifique la estructura. Lo que se quiere es que el generador, mientras hace su trabajo, *garantice* que la estructura es correcta. [1]

[1] Como sugerencia para el desarrollo, escriba una función `isValid :: NFA ->Bool` que verifique la estructura de un `NFA` cualquiera, y apóyese en ella para refinar la calidad de su generador. La función `isValid` **no** forma parte de la entrega.

Simulador de $\lambda-$NFA

Recordará de CI-3725 que los $\lambda-$NFA son equivalentes a los Autómatas Determinísticos, en virtud del algoritmo que simula el paso *simultáneo* por todos los estados válidos para cada transición. Le sugiero buscar el Sudkamp y sus notas de clase para reforzar el tema, aún cuando iremos desarrollando la solución paso a paso.

En primer lugar, es necesario identificar si un movimiento es convencional o es una $\lambda-$transición, así que debe proveer las funciones [2]

```
isMove, isLambda :: Transition -> Bool
isMove                                = not . isLambda
isLambda Lambda { from = _ , to = _ } = True
isLambda _                            = False

m1 = Move   {from = Node 0 , to = Node 1 , sym = 'a'}
m2 = Lambda {from = Node 0 , to = Node 0}
```

En el algoritmo de simulación, la manera de desplazarse a partir de un estado particular consiste en considerar la $\lambda-$clausura del estado. Luego, para cada uno de los estados, determinar a su vez aquellos estados a los cuales es posible moverse consumiendo exactamente un símbolo de entrada. Finalmente, considerar la $\lambda-$clausura de esos estados, obteniéndose así los estados destino.

Para resolver ese problema, Ud. deberá implantar varias funciones auxiliares:

- Dado un $\lambda-$NFA y un estado, calcular el conjunto de estados a los cuales se puede alcanzar con *una* $\lambda-$transición.

```
lambdaMoves :: NFA -> NFANode -> DS.Set NFANode
lambdaMoves nfa node = DS.foldl' f DS.empty (moves nfa)
  where f acc trans = if (isLambda trans) && (from trans == node)
                      then DS.insert (to trans) acc
                      else acc
```

- Dado un $\lambda-$NFA, un caracter del alfabeto y un estado, calcular el conjunto de estados a los cuales se puede alcanzar con una transición que consuma el caracter de entrada.

[2] Note que una es la negación de la otra. Implante la que le resulte "más obvia" y exprese la otra como negación de la primera en estilo *point-free* – elegancia y categoría.

```
normalMoves :: NFA -> Char -> NFANode -> DS.Set NFANode
normalMoves nfa c node = DS.foldl' f DS.empty (moves nfa)
  where f acc trans = if (isMove trans) && (from trans == node)
                                        && (sym trans  == c)
                      then DS.insert (to trans) acc
                      else acc
```

- Dado un $\lambda-$NFA, un caracter del alfabeto y un estado, calcular el conjunto de estados a los cuales se puede alcanzar consumiendo el caracter de entrada. Esta es la función que debe calcular la $\lambda-$clausura del estado inicial, los desplazamientos desde ese conjunto ahora consumiendo la entrada, y luego la $\lambda-$clausura final.

```
destinations :: NFA -> Char -> NFANode -> DS.Set NFANode
destinations nfa c node = DS.foldl g DS.empty $
                              fixSet lambdaN (DS.singleton node)
  where lambdaN n   = lambdaMoves nfa n
        g acc x     = acc `DS.union` hard x
        hard   nodo =  (before nodo) `DS.union` (after nodo)
        before nodo = DS.foldl' g1 DS.empty $
                          fixSet lambdaN (DS.singleton nodo)
           where g1 acc x = acc `DS.union` (normalMoves nfa c x)
        after  nodo = fixSet lambdaN (normalMoves nfa c nodo)
```

- Recordará que el cálculo de la $\lambda-$clausura es un algoritmo de la familia de Algoritmos de Punto Fijo. En estos algoritmos se cuenta con un conjunto inicial, una función aplicable sobre elementos del conjunto que produce nuevos conjuntos, los cuales deben ser unidos para ampliar el conjunto, hasta que no se pueda ampliar más. Escriba la función

```
fixSet :: Ord a => (a -> DS.Set a) -> DS.Set a -> DS.Set a
fixSet f s = let newSet = DS.foldl' g s s
             in  if (newSet == s)
                 then newSet
                 else fixSet f newSet
  where g acc x = DS.union acc (f x)
```

  Que es capaz de calcular el punto fijo de aplicar `f` sobre el conjunto `s`. La función `fixSet` será necesaria para poder implantar `destinations`.

Una vez implantadas estas funciones, estará en posición de implantar el simulador monádico de $\lambda-$NFA poniendo en práctica monads y transforma-

dores.

La función principal de este simulador será

```
runNFA :: NFA -> [Char] -> IO ()
runNFA nfa word = do ((a,b), c) <- evalM5 nfa (initialState word) eval5'
                     case a of
                         Left a  -> print a
                         Right a -> print b
```

que para un `nfa` particular simulará el procesamiento de la palabra de entrada `word`. El comportamiento de la función dependerá de lo que ocurra en la simulación. Por ejemplo:

- Si la palabra es *aceptada*, debe *imprimir* en pantalla la secuencia de conjuntos de estados por la cual pasó el $\lambda-$NFA.

  ```
  ghci> runNFA nfa0 "abb"
  fromList [fromList [q0],fromList [q0,q1],
            fromList [q2],fromList [q3]]
  ```

  Aquí hay *dos* usos diferentes de `fromList`: el más externo es de un `Data.Sequence` y los más internos son de `Data.Set`.

- Si la palabra es *rechazada* porque se consumió la entrada pero el $\lambda-$NFA quedó en un estado no final, debe *imprimir* en pantalla un mensaje indicando que rechazó y en cuales estados se encontraba la simulación.

  ```
  ghci> runNFA nfa0 "ab"
  Reject (fromList [q2])
  ```

- Si la palabra es *rechazada* porque no hay transiciones posibles sobre el siguiente símbolo de entrada, debe *imprimir* en pantalla un mensaje indicando que rechazó indicando la entrada restante y el estado de estancamiento.

  ```
  ghci> runNFA nfa0 "abbbb"
  Stuck (fromList [q3]) "bb"
  ```

Para escribir el simulador Ud. necesitará combinar correctamente los monads `Reader`, `Writer`, `State` y `Error`. [3] Necesitará el monad `Reader` para llevar el NFA que se está simulando, el monad `Writer` para ir conservando los conjuntos de estados por los cuales vaya avanzando, el monad `State` para mantener la entrada restante y el conjunto de estados actuales, y el monad `Error` para poder emitir las excepciones necesarias para el rechazo.

Ambos rechazos serán manejados como excepciones, así que necesitará

```
data NFAReject = Stuck (DS.Set NFANode) String
               | Reject (DS.Set NFANode)
               deriving (Show)

instance Error NFAReject
```

Necesitará un tipo de datos que se aprovechará en el Monad `State`

```
data NFARun = NFARun { w :: String, qs :: DS.Set NFANode }
            deriving (Show,Eq)
```

Como comprenderá, no puedo darle la firma de las funciones monádicas que necesitará. Después de todo, de eso se trata esta evaluación. No obstante, le sugiero estructurar su código alrededor de las siguientes funciones:

- Una función para preparar el estado inicial de la simulación, con la palabra de entrada y fijando el estado actual en `(Node 0)`.

  ```
  initialState :: String -> NFARun
  initialState word = NFARun {w = word, qs = DS.singleton (Node 0)}
  ```

- Una función para determinar si en un conjunto de estados hay uno o más que sean estados finales de un NFA.

  ```
  accepting :: NFA -> DS.Set NFANode -> Bool
  accepting nfa set =  DS.foldl' g False set
    where g acc nodo = acc || (nodo `DS.member` (final nfa))
  ```

- Una función monádica `start` que comienza la simulación a partir del estado inicial.

[3] Puede usar el monad `RWS` o el transformador `RWST` si lo desea. No es ni más fácil, ni más difícil: sólo tiene que escribir menos firmas y paréntesis.

- Una función monádica `flow` que completa la simulación. Esta función es la que debe operar en el monad combinado y hacer avanzar el $\lambda-$NFA consumiendo la entrada. Si detecta que debe rechazar la palabra, lanza la excepción adecuada; si logra procesar todo la entrada y aceptar, debe permitir acceso al historial de movimientos.
- La función `runNFA` mencionada más arriba, debe aprovechar las funciones monádicas `start` y `flow` en el contexto de los Monads adecuados para correr la simulación, obtener los resultados e imprimir.

Hay muchas formas de implantar la simulación y las funciones auxiliares, sin embargo aplican las recomendaciones que ya hemos discutido en el pasado: aprovechar funciones de orden superior y aprovechar funciones de las librerías.

Una vez que su simulación esté operando correctamente, escriba dos propiedades QuickCheck y aproveche la instancia `Arbitrary` para comprobar:

- Todo $\lambda-$NFA que tenga un estado final en la $\lambda-$clausura de su estado inicial acepta la palabra vacía.

```
prop_acceptsemptyword :: NFA -> Property
prop_acceptsemptyword nfa = undefined
```

- Cuando un $\lambda-$NFA acepta una palabra de longitud n, el camino recorrido tiene longitud $n + 1$.

```
prop_acceptancelength :: NFA -> String -> Property
prop_acceptancelength nfa w = undefined
```

Mostraremos los pasos para ir combinando correctamente los monads `Reader`, `Writer`, `State` y `Error`.

- 1) Colocamos el Monad IO ()

```
type Eval1 a = IO a

eval1 :: NFA -> [Char] -> Seq.Seq (DS.Set NFANode )
              -> DS.Set NFANode -> Eval1 (Seq.Seq (DS.Set NFANode))
eval1 nfa []     set act = return set
eval1 nfa (x:xs) set act = eval1 nfa xs ((Seq.|>) set d) d
```

```
  where d = getDest nfa x act

getDest :: NFA -> Char -> DS.Set NFANode -> DS.Set NFANode
getDest nfa char set = DS.unions . map (destinations nfa char) $ DS.toList set

evalInicial1 :: NFA -> [Char] -> Eval1 (Seq.Seq (DS.Set NFANode))
evalInicial1 nfa word = eval1 nfa word (Seq.singleton node0) node0
    where node0 = DS.singleton (initial nfa)

evalM1 :: Eval1 a -> IO (a)
evalM1 e = e
```

- 2) Colocamos el transformador de error sobre el monad IO

```
type Eval2 a =  ErrorT NFAReject IO a

eval2 :: NFA -> [Char] -> Seq.Seq (DS.Set NFANode)
             -> DS.Set NFANode -> Eval2 (Seq.Seq (DS.Set NFANode))
eval2 nfa []     set act = checkReject nfa set act
eval2 nfa w@(x:xs) set act = if (DS.null d)
                             then throwError $ Stuck act w
                             else eval2 nfa xs ((Seq.|>) set d) d
  where d = getDest nfa x act

checkReject nfa set act = if (accepting nfa act)
                          then return set
                          else throwError $ Reject act

evalInitial2 :: NFA -> [Char] -> Eval2 (Seq.Seq (DS.Set NFANode))
evalInitial2 nfa word = eval2 nfa word (Seq.singleton node0) node0
    where node0 = DS.singleton (initial nfa)

evalM2 :: Eval2 (Seq.Seq (DS.Set NFANode))
          -> IO (Either NFAReject (Seq.Seq (DS.Set NFANode)))
evalM2 = runErrorT
```

- 3) Colocamos el Monad Reader sobre el monad construido en el paso 2.

```
type Eval3 a = ReaderT NFA (ErrorT NFAReject IO) a

eval3 :: [Char] -> Seq.Seq (DS.Set NFANode)
                -> DS.Set NFANode -> Eval3 (Seq.Seq (DS.Set NFANode))
eval3 []       set act = do nfa <- ask
                            checkReject nfa set act
eval3 w@(x:xs) set act = do nfa <- ask
```

```
                            when (DS.null (d nfa)) $ throwError $ Stuck act w
                            eval3 xs ((Seq.|>) set (d nfa)) (d nfa)
  where d nfa = getDest nfa x act

evalInitial3 :: [Char] -> Eval3 (Seq.Seq (DS.Set NFANode))
evalInitial3 word = do nfa <- ask
                       eval3 word (Seq.singleton (node0 nfa)) (node0 nfa)
    where node0 nfa = DS.singleton (initial nfa)

evalM3 :: NFA -> Eval3 (Seq.Seq (DS.Set NFANode))
              -> IO (Either NFAReject (Seq.Seq (DS.Set NFANode)))
evalM3 nfa = runErrorT . (flip runReaderT) nfa
```

- 4) Agregamos el monad State a nuestra construccion. Tener encuenta que colocamos el StateT por debajo del monad ErrorT para poder llevar el estado.

```
type Eval4 a = ReaderT NFA
                  (ErrorT NFAReject
                      (StateT NFARun IO)) a

eval4 :: Seq.Seq (DS.Set NFANode) -> Eval4 (Seq.Seq (DS.Set NFANode))
eval4 set = do
   nfa <- ask
   s <- get
   let word = w s
       act  = qs s
   if (null word )
   then checkReject nfa set act
   else do let c = head word
           when (DS.null (dst nfa c act)) $
                throwError $ Stuck act word
           put $ s {w = tail word, qs = dst nfa c act}
           eval4 ((Seq.|>) set (dst nfa c act))
  where dst nfa c act = getDest nfa c act

evalInitial4 :: Eval4 (Seq.Seq (DS.Set NFANode ))
evalInitial4 = do nfa <- ask
                  eval4 (Seq.singleton (node0 nfa))
    where node0 nfa = DS.singleton (initial nfa)

evalM4 :: NFA -> NFARun
              -> Eval4 (Seq.Seq (DS.Set NFANode))
              -> IO (Either NFAReject (Seq.Seq (DS.Set NFANode )), NFARun)
evalM4 nfa init = (flip runStateT init) . runErrorT .
                  (flip runReaderT) nfa
```

- 5) Agregamos el monad Transformer WriterT para llevar una bitacora. Lo agregamos por debajo del nivel del ErrorT para que no se pierda la botacora, y por encima del transformador StateT

```
type NFALog = Seq.Seq (DS.Set NFANode)

type Eval5 a = ReaderT NFA
                  (ErrorT NFAReject
                      (WriterT NFALog
                          (StateT NFARun IO))) a

eval5' :: Eval5 ()
eval5' = do tell $ Seq.singleton (DS.singleton (Node 0))
            eval5

eval5 :: Eval5 ()
eval5 = do
   nfa <- ask
   s <- get
   let word = w s
       act  = qs s
   if (null word )
   then do unless (accepting nfa act) $ throwError $ Reject act
   else do let c = head word
           when (DS.null (dst nfa c act)) $
              throwError $ Stuck act word
           put $ s {w = tail word, qs = dst nfa c act}
           tell $ (Seq.singleton (dst nfa c act))
           eval5
  where dst nfa c act = getDest nfa c act

evalM5 :: NFA -> NFARun
              -> Eval5 ()
              -> IO ((Either NFAReject (), NFALog), NFARun)
evalM5 nfa init = (flip runStateT init) . runWriterT .
                   runErrorT . (flip runReaderT) nfa
```

Otro Beta

La vida es dura. Todos los días hay que salir a la calle, buscando la fuerza para alcanzar otro beta y echar pa'lante.

```
data Otro a = Otro ((a -> Beta) -> Beta)
```

Y se hace más difícil, porque el `Beta` está en una chamba o en hacer llave con un convive, así hasta que te toque el quieto.

```
data Beta = Chamba (IO Beta)
          | Convive Beta Beta
          | Quieto
```

Se complica ver el `Beta`, porque `IO` esconde lo que tiene. Hay que inventarse una ahí para tener idea...

```
instance Show Beta where
   show (Chamba x)    = " chamba "
   show (Convive x y) = " convive(" ++ show x
                                    ++ ","
                                    ++ show y ++ ") "
   show Quieto        = " quieto "
```

A veces hay suerte, y uno encuentra algo que hacer. Uno llega con fuerza, hace lo que tiene que hacer, y allí sigues buscando otro Beta

```
hacer :: Otro a -> Beta
hacer = undefined
```

pero es triste ver cuando simplemente es un quieto. No importa si traes fuerza, te quedas quieto.

```
quieto :: Otro a
quieto = undefined
```

Pero hay que ser positivo. Hay que pensar que si uno encuentra un oficio, uno chambea. Sólo hay que darle a la chamba y de allí uno saca fuerza

```
chambea :: IO a -> Otro a
chambea = undefined
```

y si el trabajo se complica, lo mejor es encontrar un convive para compartir la fuerza, aunque al final quedes tablas

```
convive :: Otro a -> Otro ()
convive = undefined
```

Para llegar lejos, es mejor cuadrar con un pana. Cada uno busca por su lado, y luego se juntan.

```
pana :: Otro a -> Otro a -> Otro a
pana = undefined
```

y así al final, cuando se junten los panas, hacen una vaca y se la vacilan

```
vaca :: [Beta] -> IO ()
vaca = undefined
```

Me pasaron el dato, que buscar el beta es como un perol, que con cada mano que le metes, siempre te hace echar pa'lante. Que consulte al chamo de sistemas, pa'que me muestre como hacer. Porque a esos peroles con cosas, que paso a paso avanzan, dizque los mentan "Monads". A mi no me dá el güiro, pero espero que ti si, menor.

```
instance Monad Otro where
  return x       = undefined
  (Otro f) >>= g = undefined
```

Nota: el propósito de este ejercicio es que noten que son los *tipos* los que deben describir el comportamiento. Hay sólo una manera correcta de escribir todas las funciones aquí solicitades, para lo cual lo único necesario es considerar el tipo a producir y los tipos de los argumentos. Así mismo, para escribir la instancia `Monad`, sólo es necesario respetar las firmas.
Para saber si su código funciona, primero debe compilar todo sin errores. Luego, ejecute

```
ghci> quedo cartel
```

y el contenido le hará entender si lo logró.